OO(A,D,P(c++))

In my last two articles, I talked about abstract classes, pure virtual functions and collaboration. In this article I will use both these concepts in a case study which explores the topic of Object-Oriented Analysis and Design. Although the subjects of OOA and OOD are steeped in theory and current controversy, this article will bypass all that; and present them from a pragmatic, "how-to" point of view.

Make no mistake, OOA and OOD are *big* topics. It is not feasible to cover them in any depth within the context of a single article. Therefore this article will present the basics of OOAD, and will not attempt a detailed account. However, the information in this article should be sufficient to give you an inkling or two about what OOA and OOD are all about.

There seems to be a great deal of discussion about the merits of various object oriented languages. Some of the participants in these discussions favor C++ as an OOPL based on its "nearness" to C and its pragmatic outlook. Others berate C++ as an ugly and incomplete language because of its "nearness" to C and its pragmatic outlook. I would like to make a different point altogether: The quality of an object oriented application is determined far more by the quality of its analysis and design, than by the features of the implementation language.

Certainly I believe that language features are important. Indeed, certain features such as inheritance and dynamic binding are essential to the implementation of an object-oriented design. But given that base set of features, and a reasonable amount of expressive ability, the success of an application depends on a correct analysis and design, not on the nuances of the language. In my opinion, C++ offers features in sufficient quantity to implement any object-oriented design. This, and its "nearness" to C and its pragmatic outlook, make is a natural choice for me.

If the truly important part of creating an object-oriented application is its analysis and design, it might be a good idea if we spent some time learning what object-oriented analysis and design really are. There seems to be a great deal of confusion on this subject. For example, I hear the following questions being asked quite often:

- Are there any differences between OOA and OOD? If so, what are they?
- Should we do OOA first, followed by OOD?
- What is a reasonable output from OOA, OOD?
- Are there different notations for OOA and OOD?

Before we can understand Object-Oriented Analysis well enough to answer these questions, we should first try to understand what the concept behind *Analysis* is. Webster supplies us with several interesting definitions: *"separation or breaking up of a whole into its fundamental elements or component parts."* or *"A detailed examination of anything complex."* These definitions come close to the traditional meaning of Systems Analysis, but another of the definitions rounds the concept out: *"The practice of proving a mathematical proposition by assuming the result and reasoning back to the data or to already established principles."*

Analysis, is the decomposition of an application into its constituent parts. It is the exposure of the organization of the innards of a problem. This is accomplished by beginning with a set of stated requirements, and then reasoning backward from those requirements to a set of already established software components and structures. It is a way of re-expressing a human problem in a language more suited for describing a computer application.

It is important to understand, in general terms, just what we must analyze. We analyze requirements. What are requirements? Regardless of their form, requirements boil down to a set of desired behaviors. Requirements are the behaviors that the completed application should express.

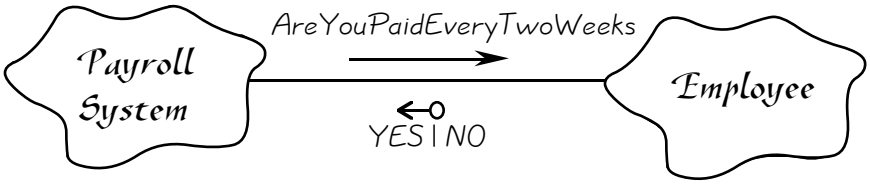
This is a central concept. Analysis is the restatement of a set of required behaviors in terms suitable for describing a computer application. This restatement is arrived at by considering the required behaviors and reasoning backwards to the components of those behaviors which can be expressed in a computer program. In the case of Object-Oriented Analysis the components of the required behaviors are objects and their collaborations.

In Object-Oriented Analysis, we start from the required behaviors and we reason backwards to find the abstractions that underlie those behaviors. Then we attempt to define the objects that represent those abstractions, and the messages which those objects pass between each other in order to implement the required behaviors. We do this by studying use cases and their scenarios. We examine each requirement, one by one, and develop dynamic scenarios composed of objects and messages which address those requirements. Booch's object diagrams are ideal for this purpose.

For example, consider a Payroll application with the following requirements:

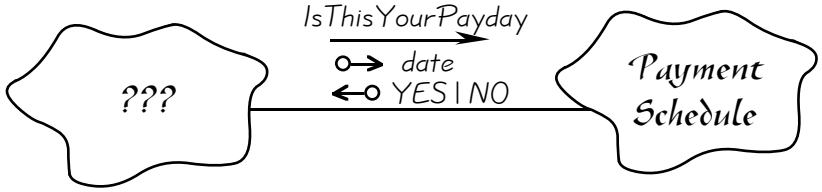
- Some employees are paid once every two weeks, while others are paid once per month.
- Salesmen are paid a base rate plus a commission on receipts.
- Other exempt employees are paid a fixed salary per pay period.
- Hourly employees are paid an hourly rate plus any authorized overtime; based on the data in their timecards.
- Employees may choose whether they will be issued a paycheck, or have their pay directly deposited into their accounts.

Lets take the first point. It appears that employees are not all paid at the same time. Some get their pay every month while others get their pay every two weeks. What, as analysts, can we say about this situation? As Object-Oriented analysts we hunt for abstractions, for objects which express the underlying concept. It would be incredibly näive of us to presume that this requirement simply implied the following:



Rather, we want to look for the motivation which underlies the requirement. What is it about this requirement that could change, and what will always be invariant. Clearly the invariant portion of the requirement is that *all employees will be paid according to some schedule.* But what could easily change is the nature of that schedule. It is not hard for us to imagine that new policies might allow for employees who are paid weekly, or even quarterly. Thus we can identify that the payment schedule for an employee is an abstraction. Although each employee has one, there may be many different kinds.

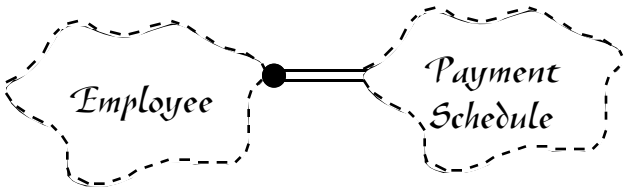
Having created the abstraction of a payment schedule, we can create our first analysis object. We will name it: *PaymentSchedule*. This object has the following interface:



This is certainly not the only adequate interface for the *PaymentSchedule* object; nor perhaps,

is it the best. But it states the abstraction succinctly enough for my tastes.

How is an *Employee* object associated with the appropriate *PaymentSchedule* object? The simplest answer is that it contains it. We can express this in a class diagram as a class relationship; as follows.



How does *PaymentSchedule* work? There are any number of ways.; lets look at one of the worst ways first. It could be a simple object with an `enum` variable inside it:

```
class PaymentSchedule
{
  public:
    enum PaymentScheduleType {biweekly, monthly};

    PaymentSchedule(const PaymentScheduleType& x)
    : itsScheduleType(x)
    {}

    int IsThisPayday(const Date&) const;

  private:
    PaymentScheduleType itsScheduleType;
};
```

It is easy to see how this class works. It is constructed with a schedule type as in:

```
PaymentSchedule schedule(PaymentSchedule::monthly);
```

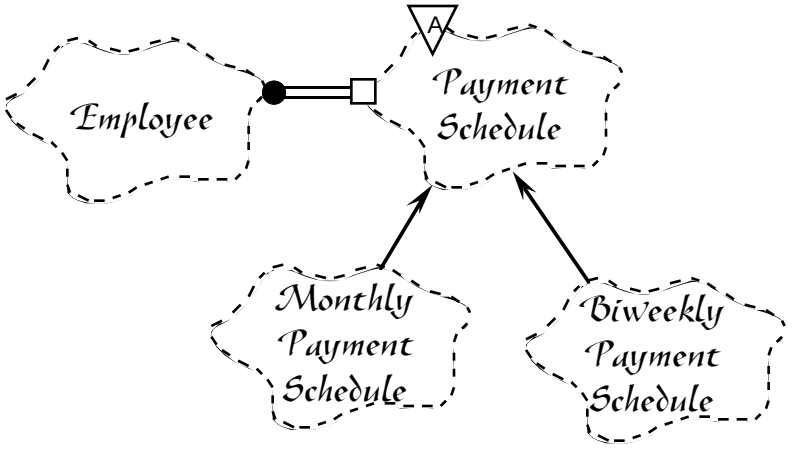
Then the object simply consults its private variable whenever it is asked whether any particular date is a payday. The method must determine if the date matches one of the enumerated schedule types.

This works, but it is a horribly icky design. I'll point out why this design is so bad in just a minute, but did you notice that I said *design*? This entire discussion of how the *PaymentSchedule* object works is a design issue. The really rotten design that I just cooked up above is completely consistent with the analysis model. Moreover it meets the requirements flawlessly. Yet it is just one of many ways that the analysis model can be satisfied, and some of those other ways are much better.

This demonstrates the point that a good analysis can be coupled with a bad design. It is not sufficient simply to recognize the abstractions in application. Effort must be applied to finding elegant designs for those abstractions.

Notice how easy and natural it was to slide from analysis into design? This is typical, and should not be avoided. We constantly shuttle back and forth across the analysis-design border while we develop object-oriented software. This shuttling is a good thing and is to be encouraged. It solidifies our faith in the analysis when we can design a proper solution. Also, if it is difficult to complete a design from the analysis model, it indicates that something may be wrong with that model, and reanalysis is indicated.

So, what was wrong with the design of *PaymentSchedule*. It is too limited and inflexible. If a new payment schedule is required, I must modify the old *PaymentSchedule* class. This risks that, during my modification, I will break the code that already exists in that class. It also puts me in the unhappy situation of having some horrible `switch` or `if/else` monstrosity in the `IsThisPayday` method. A better way to design this class is as follows:



Now we see that *PaymentSchedule* is an abstract class with at least two derivatives, one which implements the monthly payment schedule, and the other which pays employees every two weeks. Moreover, we have denoted that *Employee* contains its instance of *PaymentSchedule* by reference, in order that the containment can be polymorphic. The class declarations look like this:

```
class Employee
{
  public:

  private:
    PaymentSchedule* itsPaymentSchedule;
};

class PaymentSchedule
{
  public:
    virtual int IsThisPayday(const Date&) const = 0;
};

class MonthlyPaymentSchedule : public PaymentSchedule
{
  public:
    virtual int IsThisPayday(const Date&) const;
};

class BiweeklyPaymentSchedule : public PaymentSchedule
{
  public:
    virtual int IsThisPayday(const Date&) const;
};
```

This design allows us to add new payment schedules as needed. These payment schedules can

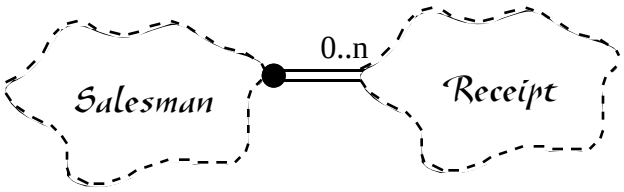
be as complex or as simple as desired. No funny `switch` or `if/else` segments are implied, and the code in previously working derivatives of `PaymentSchedule` will not be affected.

Also, this design *hides* the complexity of the concept of payment schedules. The differences between the payment schedules are ignored by all but the classes derived from *PaymentSchedule*. This is an important concept. Remember that we said that one of the goals of analysis is to expose the components of the required behaviors? In contrast, it is one of the goals of the design to *hide* the complexity of those behaviors from other parts of the design. So, analysis *exposes* while design *hides*.

Switching back to analysis now, lets look at the next requirement:

- Salesmen are paid a base rate plus a commission on receipts.

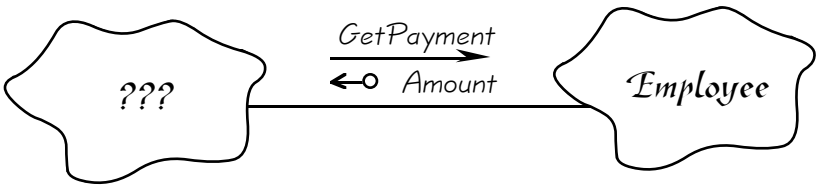
This would seem to imply that there should be a *Salesman* object which contains a set of *Receipt* objects; as follows:



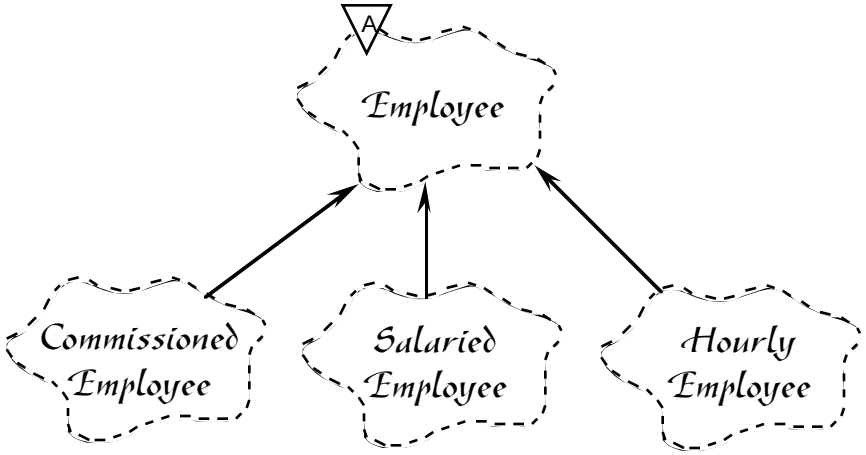
This certainly is in keeping with the requirements, but does this express the underlying motivation of the requiment? Once again, we can test this by asking what can change about the requirement, and what is invariant. This requirement is stating a method for calculating pay. In fact there are two other requirements which do exactly the same thing for different types of employees:

- Other exempt employees are paid a fixed salary per pay period.
- Hourly employees are paid an hourly rate plus any authorized overtime; based on the data in their timecards.

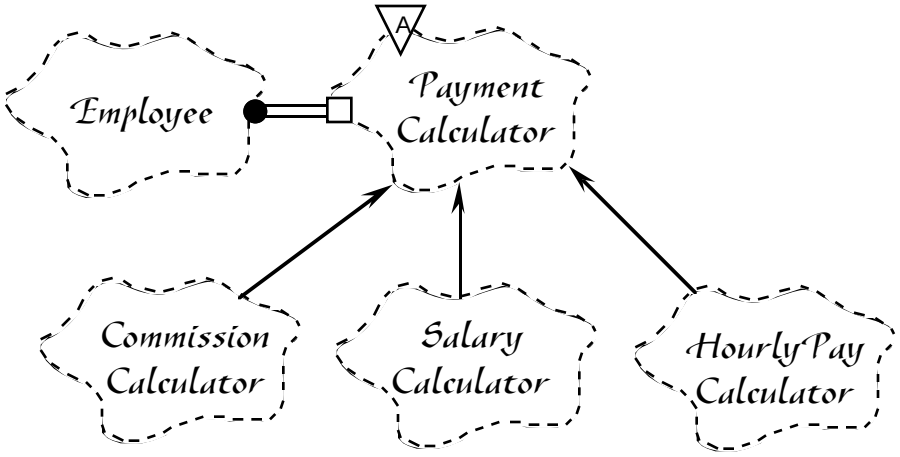
It seems clear that the underlying abstraction for these three requirements is: *There are many different types of employee, and each has its own distinct way of calculating its pay*. This leads us immediately to an interface for *Employee* objects which looks like this:



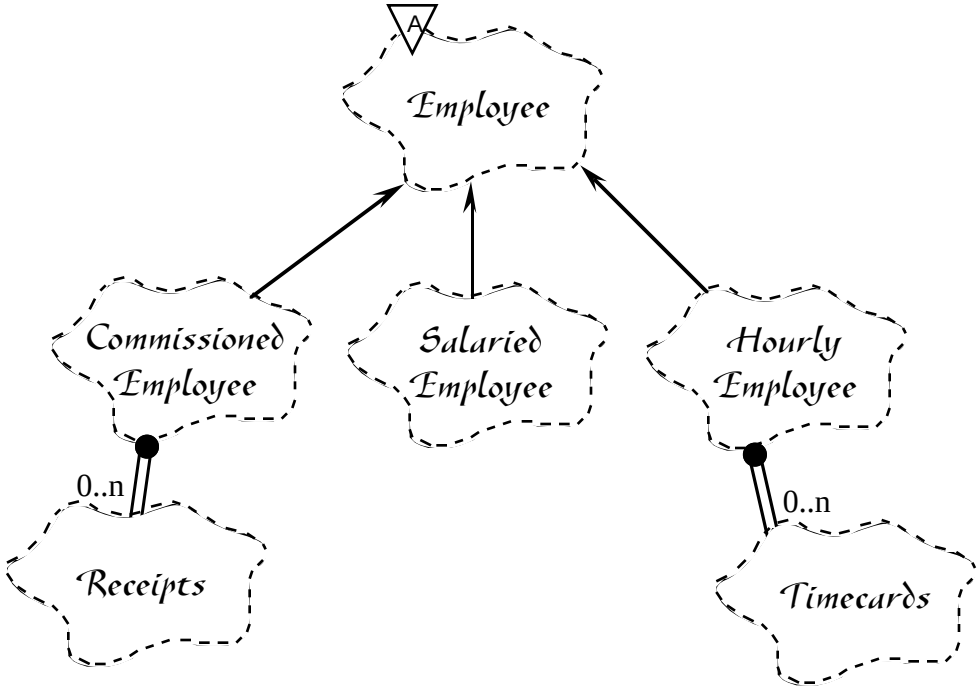
Now we are presented with several design possibilities. First, we could create many subclasses of *Employee* as follows:



Alternatively, we could have the *Employee* object contain an object called: *PaymentCalculator*. like this:



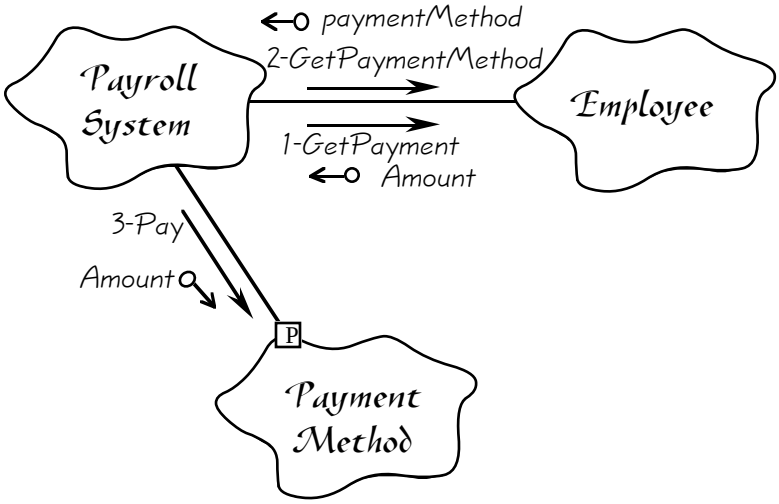
I prefer the first option becuase the calculator objects are going to need a lot of information from the *Employee* objects, and so the interface between them is going to be wide and complex. Moreover, the three different payment calculators are going to need three different types of documents to calculate pay from. The *CommissionCalculator* is going to need to see receipts. The *HourlyPayCalculator* is going to need to see time cards. The calculators can only get these documents from the *Employee* object itself. Thus there must be three different flavors of *Employee* just to hold the three different kinds of documents. So I prefer the first model, fleshed out as follows:



Finally, we turn back to the analysis of the final requirement.

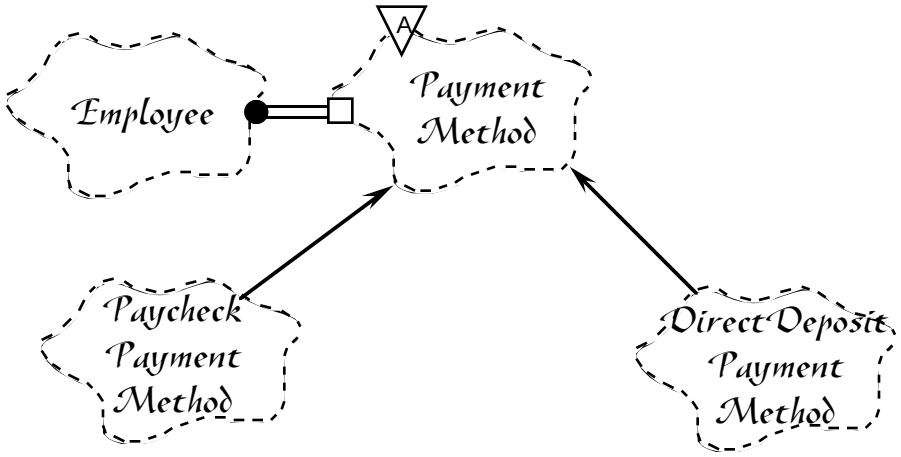
- Employees may choose whether they will be issued a paycheck, or have their pay directly deposited into their accounts.

Once again we want to find the central abstraction which motivates this requirement. And it seems clear that this abstraction is: *Every employee will be able to choose some mechanisms for being paid*. The requirement names two such mechanisms; specifically: paycheck or direct-deposit. However it seems likely that other payment methods will evolve over time. Thus, the following interface suggests itself:



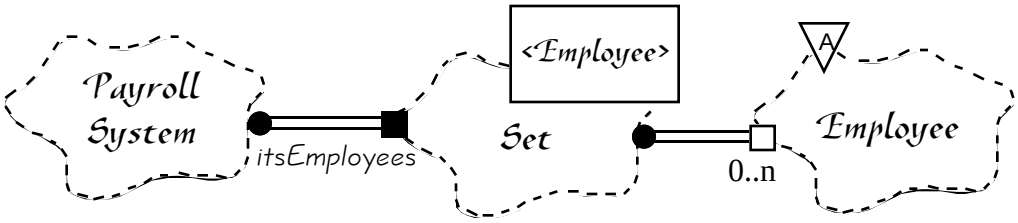
Here the *PayrollSystem* gets the payment amount and the payment method from the *Employee*. We presume that it doctors the payment amount by making income tax and FICA deductions, insurance payments, etc. And then it sends the final payment amount to the *PaymentMethod* object that it got from the *Employee*.

Given this interface, the design is as straightforward as the others have been.



Note again that *Employee* contains *PaymentMethod* by reference. This is to insure that the actual object contained can be a derivative of *PaymentMethod* and so act polymorphically.

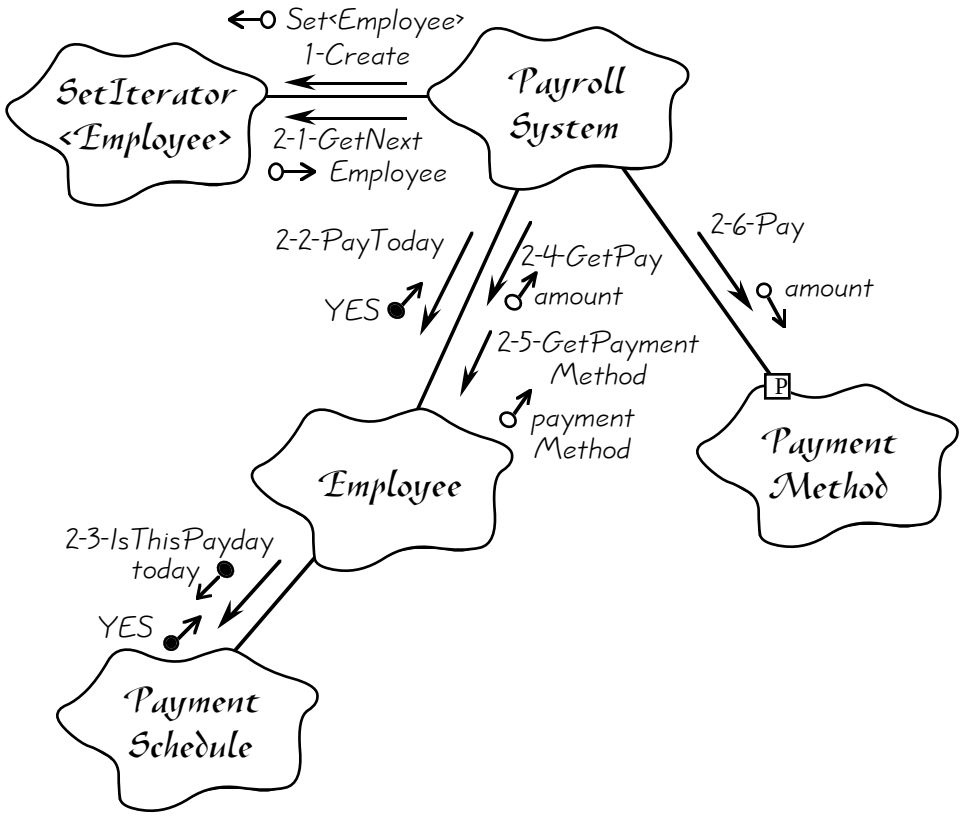
On several of the diagrams we have seen an object called: *PayrollSystem*. What is this object, and what are its class relationships? I think that this class provides the central control for the Payroll application and contains the set of all employees; as follows:



Note the "Instantiated Class" icon for *Set<Employee>*. This is a standard Booch icon, and it corresponds to the use of templates in C++. Thus the corresponding code is:

```
class PayrollSystem
{
  private:
    Set<Employee> itsEmployees;
};
```

Now we can create an object diagram which shows the gross flow of control within the entire application.



This shows just one of the many scenarios that apply to this simple application. This scenario shows the creation of the *SetIterator*, the test to see if the *Employee* should be paid today, with an affirmative result. It also shows the acquisition of the payment method, and the act of payment. Other scenarios that should be drawn would show the end of the iteration, negative responses to the *IsThisPayday* message and the flow of messages in the derivatives of *PaymentSchedule*, *PaymentMethod* and *Employee*.

Note the simplicity of this algorithm. It knows nothing of the different kinds of schedules, employees or payment methods. It simply invokes the abstractions without knowing or caring what they actually do; the hallmark of OO. The algorithm can be expressed in C++ as follows:

```
class PayrollSystem
{
  public:
    void Run();
  private:
    Set<Employee> itsEmployees;
};

void PayrollSystem::Run()
{
    SetIterator<Employee> i(itsEmployees);
    Employee* e = 0;
    while ((e = i.GetNext()) != 0)
    {
        if (e->PayToday())
        {
```

```
            double amount = e->GetPay();
            // Calculate deductions et. al.
            e->GetPaymentMethod().Pay(amount);
        }
    }
}
```

This completes our brief case study of Object-Oriented Analysis and Design. We are now in a position to re-visit the questions that appeared at the beginning of this article.

- Are there any differences between OOA and OOD? If so, what are they?

  Yes, there are differences. OOA is the act of determining the abstractions which unerly the requirements. It is the backwards reasoning that starts with the requirements and ends up with a set of objects that express the abstractions; and messages that support the required behaviors. OOD embodies the set of decisions that determines how those objects and methods will be implemented. Typically class inheritance and composition hierarchies are among those design decisions.

  OOA exposes the components of the required behaviors, in order that they may be implemented. On the other hand, OOD hides the details of the implementation so that those details do not pollute the rest of the design.

  Despite the differences, however, OOA and OOD are simply at opposite ends of the same activity. Moreover, there is no good dividing line which separates them, either in time or in function. It is better to view OOA and OOD like the different colors in a spectrum. They are certainly different, but it is difficult to tell when one starts and the other leaves off. OOAD is a continuum of activity which incorporates both analysis and design.

- Should we do OOA first, followed by OOD?

  No, they are best done concurrently. Analysis and Design should not be separated into broad phases. Nor should the analysis model be frozen before the design modeling begins. It should be evident from our case study that OOA and OOD are very closely related, and that they cooperate synergistically. The analysis model cannot be completed in the absence of the design model. And the design model cannot be completed in the absence of the analysis model. One of the most important aspects of OOAD is the synergy between the two concepts.

- What is a reasonable output from OOA, OOD?

  At very least a set of class and object diagrams. The class diagrams show the static structure of the application. It is from these diagrams that class headers can be written. Object diagrams show the dynamics of the collaborations between objects. It is from these diagrams that the class implementations will be written.

  However, in large projects these meager beginnings are hardly adequate. There is much that we have not covered in this simple case study. We have not discussed how to organize the analysis and design of a large program. How to control large scale visibility.

How to control the physical model so that a large project can be developed and released in an organized and well managed way. These topics are important parts of the OOAD universe, and should not be neglected.

- Are there different notations for OOA and OOD?

  Yes and No. OOA, since it is based on the analysis of required behaviors, tends to focus on the behavioral model. Thus tit places more emphasis on the generation of object diagrams. We saw this in the case study. In every case, the central abstraction for each requirement was documented, first, by an Object Diagram. Conversely, since OOD focuses on finding static structures which support the required dynamics, it tends to focus on class diagrams. However, there are certainly plenty of cases where class diagrams are produced by OOA activities and object diagrams are produced by OOD activities.

  One of the great advantages of OOAD over other analysis and design techniques is that the notation for analysis is compatible with the notation for design. In fact, only one set of documents is maintained. We do not separate them into "analysis documents" and "design documents".

There is one more point that I would like to make. In this case study, we walked through the requirements point by point. We analyzed each point, and tried to find the underlying abstractions. We used object diagrams to document the way that those abstractions worked. Then we used class diagrams and more object diagrams to express a design which implemented the abstractions. Thus we created a trail of documents that lead from requirements to design. Each design decision can be traced backwards, through the documentation trail, to one or more requirements.

This is a very powerful mechanism. It means that we *know* that our design will implement all the requirement. We can prove that each requirement is addressed by the design. This is in contrast to other analysis and design methodologies in which tracing from requirements to design is very difficult if not completely impossible.

Let me say this a different way. In previous analysis and design techniques, analysis focused on the static nature of the application as opposed to its dynamics. Rather than beginning with a point by point analysis of the requirements, a high level picture was drawn which was supposed to represent the amalgamation of all the requirements. Then this picture was decomposed into finer and finer partitions, each of which were supposed to express one or more requirements. When you were done with this kind of analysis, you had a set of pretty pictures, but did you have any proof that your partitioning was correct and that your pictures really represented an analysis of the requirements? Typically not.

However, in this case study, by employing the techniques of OOA and OOD, we have provided an analysis and design of each requirement. We analyzed each requirement by considering the scenarios and use cases which contained the required behavior. Then we designed the software structures which would best implement those scenarios. We know that our design supports our analysis, and we know that our analysis is a set of abstractions which are based on the requirements. Therefore we can prove that each requirement is satisfied by the design; and that is an enormously powerful ability.

In this article we have used a case study to learn about how Object-Oriented Analysis and Design are employed in the creation of a simple application. We saw how to analyze a requirement by searching for the underlying abstraction, and then expressing that abstraction as a collaboration

between objects. We also saw how to design the class structures which most effectively support those collaborations.

Although this brief article does not do justice to the complex topic of OOAD, it is my hope that it has served as an appropriate introduction. If so, you should now be aware of the power and expressiveness of OOAD. It is my opinion that proper use of the object paradigm is embodied in the practice of OOAD, and that the object model cannot be properly exploited in any other manner. The most important decision a software engineer will make with regard to developing an application is whether he will spend the time needed in analyzing and designing it.